PHASE1-1.5 PART1: LangGraph Setup for Cost Agent - Code Implementation

Document Version: 1.0
Date: October 21, 2025
Phase: Foundation - Week 1
Status: Ready for Implementation
Prerequisites: PHASE1-1.1 (Cost Agent Skeleton) completed

📋 TABLE OF CONTENTS

🎯 OVERVIEW

Purpose

This phase sets up LangGraph as the workflow orchestration engine for the Cost Agent. LangGraph provides:

- **State management** for complex optimization workflows
- **Conditional routing** for decision-based execution
- **Memory/checkpointing** for workflow persistence
- **Human-in-the-loop** for approval workflows

Time Estimate

- **Code Generation:** 30 minutes
- **Verification:** 25 minutes
- **Total:** ~55 minutes

Success Criteria

✅ LangGraph properly configured with state machines
✅ Basic workflow executes end-to-end
✅ State persistence works (PostgreSQL checkpointing)
✅ Can demonstrate conditional routing
✅ Tests pass with 80%+ coverage

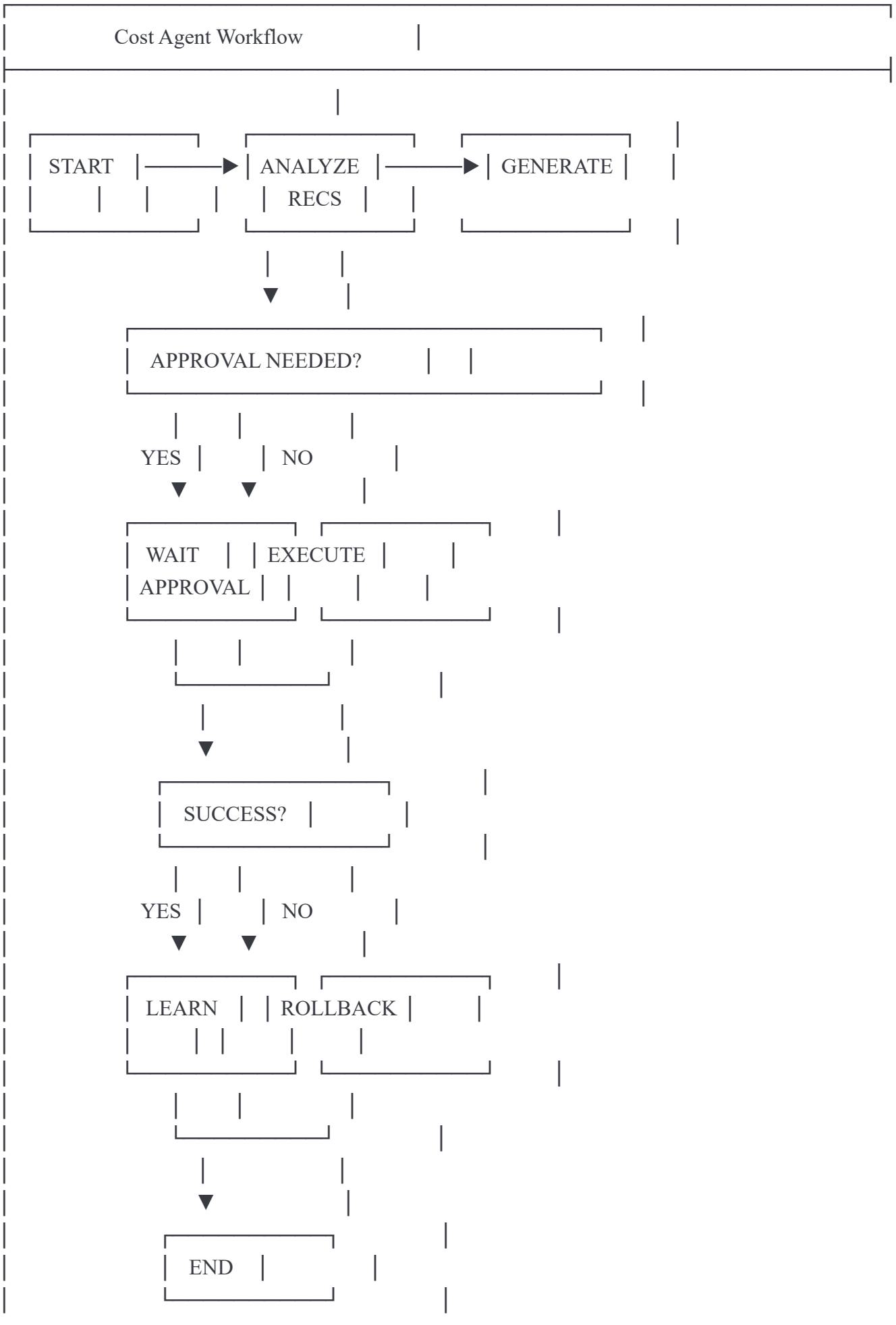
🏗️ WHAT YOU'LL BUILD

Core Components

1. **State Definitions** (`src/workflows/states.py`)
   - `OptimizationState`: Base state for all workflows
   - `SpotMigrationState`: Spot instance migration state
   - Type-safe state management
2. **Base Workflow** (`src/workflows/base.py`)
   - Abstract workflow class
   - Common patterns (approval, rollback, learning)
   - Reusable nodes and edges
3. **Graph Builder** (`src/workflows/graph_builder.py`)
   - StateGraph construction
   - Checkpointing integration
   - Conditional edge routing
4. **PostgreSQL Checkpointer** (`src/workflows/checkpointer.py`)
   - State persistence
   - Workflow resume capability
   - Audit trail

🏛️ ARCHITECTURE

LangGraph Workflow Pattern


Cost Agent Workflow
START
ANALYZE
RECS
GENERATE
APPROVAL NEEDED?
YES
NO
WAIT
APPROVAL
EXECUTE
SUCCESS?
YES
NO
LEARN
ROLLBACK
END

State Flow

python

```python
class OptimizationState(TypedDict):
    """Base state for all optimization workflows"""

    # Input
    customer_id: str
    infrastructure: Dict[str, Any]
    current_costs: Dict[str, float]

    # Analysis
    analysis_results: Optional[Dict[str, Any]]
    recommendations: Optional[List[Dict[str, Any]]]

    # Approval
    requires_approval: bool
    approval_status: Optional[str]

    # Execution
    execution_results: Optional[Dict[str, Any]]
    success: bool

    # Learning
    outcome: Optional[Dict[str, Any]]

    # Error handling
    errors: List[str]
    rollback_needed: bool
```

📦 DEPENDENCIES

Python Packages

Add to `requirements.txt`:

txt

```
# Existing dependencies
fastapi==0.104.1
uvicorn[standard]==0.24.0
pydantic==2.5.0
pydantic-settings==2.1.0
sqlalchemy==2.0.23
asyncpg==0.29.0
redis==5.0.1

# NEW: LangGraph dependencies
langgraph==0.0.40
langchain==0.1.0
langchain-core==0.1.10
langchain-openai==0.0.2
psycopg2-binary==2.9.9

# Testing
pytest==7.4.3
pytest-asyncio==0.21.1
pytest-cov==4.1.0
httpx==0.25.2
```

Installation

bash

```
cd services/cost-agent
pip install -r requirements.txt --break-system-packages
```

🔧 IMPLEMENTATION STEPS

Step 1: Create State Definitions

File: `services/cost-agent/src/workflows/states.py`

python

```
"""
State definitions for Cost Agent workflows using LangGraph.
Each workflow has its own state schema that extends the base OptimizationState.
"""

from typing import Dict, List, Optional, Any, TypedDict
from datetime import datetime


class OptimizationState(TypedDict):
    """Base state for all optimization workflows"""

    # Identification
    customer_id: str
    workflow_id: str
    workflow_type: str

    # Input data
    infrastructure: Dict[str, Any]
    current_costs: Dict[str, float]
    constraints: Dict[str, Any]

    # Analysis phase
    analysis_results: Optional[Dict[str, Any]]

    # Recommendation phase
    recommendations: Optional[List[Dict[str, Any]]]
    estimated_savings: Optional[float]
    confidence_score: Optional[float]

    # Approval phase
    requires_approval: bool
    approval_status: Optional[str]  # "pending", "approved", "rejected"
    approval_reason: Optional[str]

    # Execution phase
    execution_results: Optional[Dict[str, Any]]
    execution_status: Optional[str]  # "running", "success", "failed"

    # Success tracking
    success: bool
```

```
    # Learning phase
    outcome: Optional[Dict[str, Any]]
    learned: bool

    # Error handling
    errors: List[str]
    rollback_needed: bool
    rollback_completed: bool

    # Metadata
    created_at: datetime
    updated_at: datetime
    metadata: Dict[str, Any]


class SpotMigrationState(OptimizationState):
    """State for spot instance migration workflow"""

    # Spot-specific fields
    instances_to_migrate: Optional[List[Dict[str, Any]]]
    spot_availability: Optional[Dict[str, Any]]
    migration_plan: Optional[Dict[str, Any]]
    fallback_strategy: Optional[str]

    # Migration execution
    migrated_instances: List[str]
    failed_instances: List[str]
    interruption_rate: Optional[float]


class ReservedInstanceState(OptimizationState):
    """State for reserved instance optimization workflow"""

    # RI-specific fields
    usage_patterns: Optional[Dict[str, Any]]
    ri_recommendations: Optional[List[Dict[str, Any]]]
    commitment_period: Optional[str]  # "1-year", "3-year"
    payment_option: Optional[str]  # "all-upfront", "partial", "no-upfront"

    # Purchase tracking
```

```
    purchased_ris: List[Dict[str, Any]]
    total_upfront_cost: Optional[float]


class RightSizingState(OptimizationState):
    """State for instance right-sizing workflow"""

    # Right-sizing specific fields
    utilization_data: Optional[Dict[str, Any]]
    resize_recommendations: Optional[List[Dict[str, Any]]]

    # Resize execution
    resized_instances: List[Dict[str, Any]]
    performance_impact: Optional[Dict[str, Any]]


def create_initial_state(
    customer_id: str,
    workflow_type: str,
    infrastructure: Dict[str, Any],
    current_costs: Dict[str, float],
    constraints: Optional[Dict[str, Any]] = None
) -> OptimizationState:
    """
    Create initial state for a workflow.

    Args:
        customer_id: Customer identifier
        workflow_type: Type of workflow ("spot_migration", "reserved_instance", etc.)
        infrastructure: Infrastructure details
        current_costs: Current cost data
        constraints: Optional constraints

    Returns:
        Initial state dictionary
    """
    import uuid

    return OptimizationState(
        # Identification
        customer_id=customer_id,
```

```
workflow_id=str(uuid.uuid4()),
workflow_type=workflow_type,

# Input
infrastructure=infrastructure,
current_costs=current_costs,
constraints=constraints or {},

# Analysis
analysis_results=None,

# Recommendations
recommendations=None,
estimated_savings=None,
confidence_score=None,

# Approval
requires_approval=True,  # Default to requiring approval
approval_status=None,
approval_reason=None,

# Execution
execution_results=None,
execution_status=None,

# Success
success=False,

# Learning
outcome=None,
learned=False,

# Errors
errors=[],
rollback_needed=False,
rollback_completed=False,

# Metadata
created_at=datetime.utcnow(),
updated_at=datetime.utcnow(),
```

```
        metadata={}
    )
```

Step 2: Create PostgreSQL Checkpointer

File: `services/cost-agent/src/workflows/checkpointer.py`

python

```
"""
PostgreSQL-backed checkpointer for LangGraph workflows.
Enables workflow persistence and resumption.
"""

from typing import Dict, Any, Optional, List
from datetime import datetime
import json
import logging

from langgraph.checkpoint import BaseCheckpointSaver
from langgraph.checkpoint.base import Checkpoint, CheckpointMetadata
from sqlalchemy import (
    Column, String, DateTime, Text, Integer,
    create_engine, MetaData, Table, select, update, insert
)
from sqlalchemy.dialects.postgresql import JSONB

logger = logging.getLogger(__name__)


class PostgreSQLCheckpointer(BaseCheckpointSaver):
    """
    Checkpoint saver that persists to PostgreSQL.
    Stores workflow state for recovery and audit trail.
    """

    def __init__(self, connection_string: str):
        """
        Initialize checkpointer with database connection.

        Args:
            connection_string: PostgreSQL connection string
        """
        self.engine = create_engine(connection_string)
        self.metadata = MetaData()

        # Define checkpoints table
        self.checkpoints = Table(
            'workflow_checkpoints',
            self.metadata,
```

```
        Column('workflow_id', String, primary_key=True),
        Column('checkpoint_id', String, primary_key=True),
        Column('parent_id', String, nullable=True),
        Column('workflow_type', String, nullable=False),
        Column('customer_id', String, nullable=False),
        Column('state', JSONB, nullable=False),
        Column('metadata', JSONB, nullable=True),
        Column('created_at', DateTime, default=datetime.utcnow),
        Column('updated_at', DateTime, default=datetime.utcnow, onupdate=datetime.utcnow),
    )

    # Create table if not exists
    self.metadata.create_all(self.engine)

def put(
    self,
    config: Dict[str, Any],
    checkpoint: Checkpoint,
    metadata: CheckpointMetadata
) -> None:
    """
    Save a checkpoint to the database.

    Args:
        config: Configuration containing workflow_id, etc.
        checkpoint: The checkpoint to save
        metadata: Metadata about the checkpoint
    """
    workflow_id = config.get("configurable", {}).get("workflow_id")
    if not workflow_id:
        raise ValueError("workflow_id required in config")

    checkpoint_id = checkpoint.get("id", f"{workflow_id}-{datetime.utcnow().isoformat()}")

    # Extract metadata
    customer_id = metadata.get("customer_id", "unknown")
    workflow_type = metadata.get("workflow_type", "unknown")
    parent_id = metadata.get("parent_id")

    # Serialize state
    state_json = json.loads(json.dumps(checkpoint, default=str))
```

```
    metadata_json = json.loads(json.dumps(metadata, default=str))

    with self.engine.begin() as conn:
        # Upsert checkpoint
        stmt = insert(self.checkpoints).values(
            workflow_id=workflow_id,
            checkpoint_id=checkpoint_id,
            parent_id=parent_id,
            workflow_type=workflow_type,
            customer_id=customer_id,
            state=state_json,
            metadata=metadata_json,
        )

        # On conflict, update
        stmt = stmt.on_conflict_do_update(
            index_elements=['workflow_id', 'checkpoint_id'],
            set_={
                'state': stmt.excluded.state,
                'metadata': stmt.excluded.metadata,
                'updated_at': datetime.utcnow(),
            }
        )

        conn.execute(stmt)

    logger.info(f"Saved checkpoint {checkpoint_id} for workflow {workflow_id}")

def get(
    self,
    config: Dict[str, Any]
) -> Optional[Checkpoint]:
    """
    Retrieve the latest checkpoint for a workflow.

    Args:
        config: Configuration containing workflow_id

    Returns:
        Latest checkpoint or None
    """
```

```
    workflow_id = config.get("configurable", {}).get("workflow_id")
    if not workflow_id:
        return None

    with self.engine.begin() as conn:
        stmt = (
            select(self.checkpoints)
            .where(self.checkpoints.c.workflow_id == workflow_id)
            .order_by(self.checkpoints.c.created_at.desc())
            .limit(1)
        )

        result = conn.execute(stmt).fetchone()

        if result:
            return result.state

        return None

def list(
    self,
    config: Dict[str, Any],
    limit: int = 10
) -> List[Checkpoint]:
    """
    List checkpoints for a workflow.

    Args:
        config: Configuration containing workflow_id
        limit: Maximum number of checkpoints to return

    Returns:
        List of checkpoints
    """
    workflow_id = config.get("configurable", {}).get("workflow_id")
    if not workflow_id:
        return []

    with self.engine.begin() as conn:
        stmt = (
            select(self.checkpoints)
```

```
        .where(self.checkpoints.c.workflow_id == workflow_id)
        .order_by(self.checkpoints.c.created_at.desc())
        .limit(limit)
    )

    results = conn.execute(stmt).fetchall()

    return [row.state for row in results]
```

Step 3: Create Base Workflow Class

File: `services/cost-agent/src/workflows/base.py`

python

```
"""
Base workflow class for Cost Agent optimizations.
Provides common patterns for analysis, recommendations, approval, and execution.
"""

from typing import Dict, Any, Optional, List
from abc import ABC, abstractmethod
import logging

from langgraph.graph import StateGraph, END
from .states import OptimizationState
from .checkpointer import PostgreSQLCheckpointer

logger = logging.getLogger(__name__)


class BaseOptimizationWorkflow(ABC):
    """
    Abstract base class for optimization workflows.
    Implements common patterns like approval, rollback, and learning.
    """

    def __init__(
        self,
        checkpointer: Optional[PostgreSQLCheckpointer] = None
    ):
        """
        Initialize workflow with optional checkpointing.

        Args:
            checkpointer: PostgreSQL checkpointer for state persistence
        """
        self.checkpointer = checkpointer
        self.graph: Optional[StateGraph] = None

    # ==========================================================
    # ABSTRACT METHODS - Must be implemented by subclasses
    # ==========================================================

    @abstractmethod
    async def analyze(self, state: OptimizationState) -> OptimizationState:
```

```
    """
    Analyze infrastructure and identify optimization opportunities.

    Args:
        state: Current workflow state

    Returns:
        Updated state with analysis results
    """
    pass

@abstractmethod
async def generate_recommendations(
    self,
    state: OptimizationState
) -> OptimizationState:
    """
    Generate specific optimization recommendations.

    Args:
        state: State with analysis results

    Returns:
        Updated state with recommendations
    """
    pass

@abstractmethod
async def execute(self, state: OptimizationState) -> OptimizationState:
    """
    Execute the approved optimizations.

    Args:
        state: State with approved recommendations

    Returns:
        Updated state with execution results
    """
    pass

# ============================================================
```

```python
# COMMON WORKFLOW NODES
# ============================================================

async def check_approval(self, state: OptimizationState) -> OptimizationState:
    """
    Check if recommendations require approval.
    Sets requires_approval flag based on risk/cost thresholds.

    Args:
        state: State with recommendations

    Returns:
        Updated state with approval requirement
    """
    recommendations = state.get("recommendations", [])
    estimated_savings = state.get("estimated_savings", 0)

    # Require approval if:
    # - High-impact changes (>$10K/month savings)
    # - Many instances affected (>10)
    # - Low confidence (<0.8)

    high_impact = estimated_savings > 10000
    many_instances = len(recommendations) > 10
    low_confidence = state.get("confidence_score", 1.0) < 0.8

    requires_approval = high_impact or many_instances or low_confidence

    state["requires_approval"] = requires_approval
    state["approval_status"] = "pending" if requires_approval else "approved"

    logger.info(
        f"Approval check: requires={requires_approval}, "
        f"impact=${estimated_savings}, instances={len(recommendations)}, "
        f"confidence={state.get('confidence_score', 1.0)}"
    )

    return state

async def wait_for_approval(self, state: OptimizationState) -> OptimizationState:
    """
```

```
    Wait for human approval.
    In production, this would integrate with approval systems.

    Args:
        state: State pending approval

    Returns:
        State with approval status
    """
    # In real system, this would:
    # 1. Send notification to customer
    # 2. Wait for approval via API/portal
    # 3. Resume workflow when approved/rejected

    logger.info(f"Waiting for approval for workflow {state['workflow_id']}")

    # For now, we'll mark as pending
    # The workflow will pause here until resumed with approval
    state["approval_status"] = "pending"

    return state

async def rollback(self, state: OptimizationState) -> OptimizationState:
    """
    Rollback changes if execution failed.

    Args:
        state: State with failed execution

    Returns:
        Updated state with rollback status
    """
    logger.warning(f"Rolling back workflow {state['workflow_id']}")

    execution_results = state.get("execution_results", {})

    # Implement rollback logic
    # This is workflow-specific, but common patterns:
    # - Revert infrastructure changes
    # - Restore previous configuration
    # - Notify customer
```

```
    state["rollback_completed"] = True
    state["errors"].append("Execution failed, rollback completed")

    logger.info(f"Rollback completed for workflow {state['workflow_id']}")

    return state

async def learn(self, state: OptimizationState) -> OptimizationState:
    """
    Learn from the outcome to improve future recommendations.

    Args:
        state: State with execution results

    Returns:
        Updated state with learning recorded
    """
    # Extract outcome metrics
    outcome = {
        "workflow_id": state["workflow_id"],
        "workflow_type": state["workflow_type"],
        "success": state["success"],
        "estimated_savings": state.get("estimated_savings"),
        "actual_savings": state.get("execution_results", {}).get("actual_savings"),
        "confidence_score": state.get("confidence_score"),
        "execution_time": state.get("execution_results", {}).get("duration_seconds"),
    }

    # Store in vector database (Qdrant) for similarity search
    # This enables learning from similar past optimizations

    state["outcome"] = outcome
    state["learned"] = True

    logger.info(f"Learning recorded for workflow {state['workflow_id']}: {outcome}")

    return state

# ============================================================
# CONDITIONAL ROUTING
```

```
    # ============================================================

    def should_wait_for_approval(self, state: OptimizationState) -> str:
        """
        Route to approval or execution based on requirements.

        Args:
            state: Current state

        Returns:
            Next node name
        """
        if state.get("requires_approval") and state.get("approval_status") == "pending":
            return "wait_approval"
        return "execute"

    def should_rollback(self, state: OptimizationState) -> str:
        """
        Route to rollback or learn based on success.

        Args:
            state: Current state

        Returns:
            Next node name
        """
        if state.get("rollback_needed"):
            return "rollback"
        return "learn"

    # ============================================================
    # GRAPH CONSTRUCTION
    # ============================================================

    def build_graph(self) -> StateGraph:
        """
        Build the LangGraph workflow.
        Subclasses can override to customize the graph.

        Returns:
            Compiled StateGraph
```

```
"""
# Create graph
graph = StateGraph(OptimizationState)

# Add nodes
graph.add_node("analyze", self.analyze)
graph.add_node("generate_recommendations", self.generate_recommendations)
graph.add_node("check_approval", self.check_approval)
graph.add_node("wait_approval", self.wait_for_approval)
graph.add_node("execute", self.execute)
graph.add_node("rollback", self.rollback)
graph.add_node("learn", self.learn)

# Add edges
graph.set_entry_point("analyze")
graph.add_edge("analyze", "generate_recommendations")
graph.add_edge("generate_recommendations", "check_approval")

# Conditional: approval needed?
graph.add_conditional_edges(
    "check_approval",
    self.should_wait_for_approval,
    {
        "wait_approval": "wait_approval",
        "execute": "execute"
    }
)

# After approval granted, execute
graph.add_edge("wait_approval", "execute")

# Conditional: success or rollback?
graph.add_conditional_edges(
    "execute",
    self.should_rollback,
    {
        "rollback": "rollback",
        "learn": "learn"
    }
)
```

```
    # Both rollback and learn end the workflow
    graph.add_edge("rollback", END)
    graph.add_edge("learn", END)

    # Compile with checkpointer
    self.graph = graph.compile(checkpointer=self.checkpointer)

    return self.graph

# ============================================================
# EXECUTION
# ============================================================

async def run(
    self,
    initial_state: OptimizationState,
    config: Optional[Dict[str, Any]] = None
) -> OptimizationState:
    """
    Run the workflow with the given initial state.

    Args:
        initial_state: Starting state
        config: Optional configuration (for checkpointing)

    Returns:
        Final state after workflow completion
    """
    if not self.graph:
        self.build_graph()

    # Run workflow
    final_state = await self.graph.ainvoke(initial_state, config=config)

    return final_state
```

Step 4: Create Graph Builder Utility

File: `services/cost-agent/src/workflows/graph_builder.py`

python

```
"""
Utility functions for building LangGraph workflows.
"""

from typing import Dict, Any, Callable
import logging

from langgraph.graph import StateGraph
from .states import OptimizationState
from .checkpointer import PostgreSQLCheckpointer

logger = logging.getLogger(__name__)


def create_simple_workflow(
    nodes: Dict[str, Callable],
    edges: list[tuple[str, str]],
    conditional_edges: list[tuple[str, Callable, Dict[str, str]]],
    entry_point: str,
    checkpointer: PostgreSQLCheckpointer = None
) -> StateGraph:
    """
    Create a simple workflow from node and edge definitions.

    Args:
        nodes: Dictionary of {node_name: node_function}
        edges: List of (from_node, to_node) tuples
        conditional_edges: List of (node, condition_func, route_map) tuples
        entry_point: Name of the starting node
        checkpointer: Optional checkpointer for state persistence

    Returns:
        Compiled StateGraph

    Example:
```python
        workflow = create_simple_workflow(
            nodes={
                "analyze": analyze_func,
                "execute": execute_func
            },
```
```

```
        edges=[("analyze", "execute")],
        conditional_edges=[],
        entry_point="analyze"
    )
```
    """
    graph = StateGraph(OptimizationState)

    # Add all nodes
    for name, func in nodes.items():
        graph.add_node(name, func)
        logger.debug(f"Added node: {name}")

    # Set entry point
    graph.set_entry_point(entry_point)
    logger.debug(f"Set entry point: {entry_point}")

    # Add regular edges
    for from_node, to_node in edges:
        graph.add_edge(from_node, to_node)
        logger.debug(f"Added edge: {from_node} -> {to_node}")

    # Add conditional edges
    for source, condition, route_map in conditional_edges:
        graph.add_conditional_edges(source, condition, route_map)
        logger.debug(f"Added conditional edge from {source}: {route_map}")

    # Compile
    compiled = graph.compile(checkpointer=checkpointer)
    logger.info(f"Workflow compiled with {len(nodes)} nodes")

    return compiled

def visualize_workflow(graph: StateGraph, output_path: str = "workflow.png"):
    """
    Generate a visualization of the workflow graph.
    Requires graphviz to be installed.

    Args:
        graph: The workflow graph

```
    output_path: Where to save the visualization
    """
    try:
        from langgraph.graph.graph import CompiledGraph

        if isinstance(graph, CompiledGraph):
            # Get the underlying graph
            dot = graph.get_graph().draw_mermaid()

            # Save as mermaid diagram
            mermaid_path = output_path.replace('.png', '.mmd')
            with open(mermaid_path, 'w') as f:
                f.write(dot)

            logger.info(f"Workflow diagram saved to {mermaid_path}")
            logger.info("To visualize, paste content into https://mermaid.live")

    except ImportError:
        logger.warning("graphviz not available, skipping visualization")
    except Exception as e:
        logger.error(f"Error generating visualization: {e}")
```

Step 5: Create Tests

File: `services/cost-agent/tests/test_workflows.py`

python

```
"""
Tests for LangGraph workflow setup.
"""

import pytest
from datetime import datetime
from unittest.mock import Mock, AsyncMock

from src.workflows.states import (
    OptimizationState,
    SpotMigrationState,
    create_initial_state
)
from src.workflows.base import BaseOptimizationWorkflow
from src.workflows.checkpointer import PostgreSQLCheckpointer


class MockOptimizationWorkflow(BaseOptimizationWorkflow):
    """Mock workflow for testing"""

    async def analyze(self, state: OptimizationState) -> OptimizationState:
        state["analysis_results"] = {"finding": "test"}
        return state

    async def generate_recommendations(self, state: OptimizationState) -> OptimizationState:
        state["recommendations"] = [{"type": "test", "savings": 1000}]
        state["estimated_savings"] = 1000
        state["confidence_score"] = 0.9
        return state

    async def execute(self, state: OptimizationState) -> OptimizationState:
        state["execution_results"] = {"status": "success"}
        state["execution_status"] = "success"
        state["success"] = True
        return state


@pytest.mark.asyncio
class TestWorkflowStates:
    """Test workflow state management"""
```

```python
def test_create_initial_state(self):
    """Test initial state creation"""
    state = create_initial_state(
        customer_id="cust_123",
        workflow_type="spot_migration",
        infrastructure={"instances": []},
        current_costs={"monthly": 10000}
    )

    assert state["customer_id"] == "cust_123"
    assert state["workflow_type"] == "spot_migration"
    assert state["requires_approval"] is True
    assert state["success"] is False
    assert len(state["errors"]) == 0

def test_spot_migration_state(self):
    """Test spot migration specific state"""
    state = SpotMigrationState(
        customer_id="cust_123",
        workflow_id="wf_123",
        workflow_type="spot_migration",
        infrastructure={},
        current_costs={},
        constraints={},

        # Spot-specific
        instances_to_migrate=[{"id": "i-123"}],
        spot_availability={"us-east-1": 0.95},
        migration_plan={"batch_size": 5},
        fallback_strategy="on-demand",
        migrated_instances=["i-123"],
        failed_instances=[],
        interruption_rate=0.02,

        # Base fields
        analysis_results=None,
        recommendations=None,
        estimated_savings=None,
        confidence_score=None,
        requires_approval=True,
        approval_status=None,
```

```
            approval_reason=None,
            execution_results=None,
            execution_status=None,
            success=False,
            outcome=None,
            learned=False,
            errors=[],
            rollback_needed=False,
            rollback_completed=False,
            created_at=datetime.utcnow(),
            updated_at=datetime.utcnow(),
            metadata={}
        )

        assert len(state["instances_to_migrate"]) == 1
        assert state["spot_availability"]["us-east-1"] == 0.95


@pytest.mark.asyncio
class TestBaseWorkflow:
    """Test base workflow functionality"""

    async def test_check_approval_high_impact(self):
        """Test approval required for high-impact changes"""
        workflow = MockOptimizationWorkflow()

        state = create_initial_state(
            customer_id="cust_123",
            workflow_type="test",
            infrastructure={},
            current_costs={}
        )
        state["recommendations"] = [{"id": 1}]
        state["estimated_savings"] = 15000  # High impact
        state["confidence_score"] = 0.9

        result = await workflow.check_approval(state)

        assert result["requires_approval"] is True
        assert result["approval_status"] == "pending"
```

```python
async def test_check_approval_low_impact(self):
    """Test auto-approval for low-impact changes"""
    workflow = MockOptimizationWorkflow()

    state = create_initial_state(
        customer_id="cust_123",
        workflow_type="test",
        infrastructure={},
        current_costs={}
    )
    state["recommendations"] = [{"id": 1}]
    state["estimated_savings"] = 500  # Low impact
    state["confidence_score"] = 0.95  # High confidence

    result = await workflow.check_approval(state)

    assert result["requires_approval"] is False
    assert result["approval_status"] == "approved"

async def test_learn_from_success(self):
    """Test learning from successful execution"""
    workflow = MockOptimizationWorkflow()

    state = create_initial_state(
        customer_id="cust_123",
        workflow_type="test",
        infrastructure={},
        current_costs={}
    )
    state["success"] = True
    state["estimated_savings"] = 5000
    state["confidence_score"] = 0.9
    state["execution_results"] = {
        "actual_savings": 5200,
        "duration_seconds": 300
    }

    result = await workflow.learn(state)

    assert result["learned"] is True
    assert result["outcome"]["success"] is True
```

```
        assert result["outcome"]["actual_savings"] == 5200

    async def test_rollback(self):
        """Test rollback functionality"""
        workflow = MockOptimizationWorkflow()

        state = create_initial_state(
            customer_id="cust_123",
            workflow_type="test",
            infrastructure={},
            current_costs={}
        )
        state["rollback_needed"] = True
        state["execution_results"] = {"error": "Something failed"}

        result = await workflow.rollback(state)

        assert result["rollback_completed"] is True
        assert len(result["errors"]) > 0


@pytest.mark.asyncio
class TestWorkflowExecution:
    """Test end-to-end workflow execution"""

    async def test_full_workflow_with_approval(self):
        """Test complete workflow requiring approval"""
        workflow = MockOptimizationWorkflow()
        workflow.build_graph()

        initial_state = create_initial_state(
            customer_id="cust_123",
            workflow_type="test",
            infrastructure={"instances": []},
            current_costs={"monthly": 10000}
        )

        # Run until approval needed
        # Note: In real system, workflow would pause here
        # For testing, we'll manually approve
        initial_state["approval_status"] = "approved"  # Simulate approval
```

```
    final_state = await workflow.run(initial_state)

    # Verify workflow completed
    assert final_state["success"] is True
    assert final_state["learned"] is True
    assert "analysis_results" in final_state
    assert "recommendations" in final_state
    assert "execution_results" in final_state


@pytest.mark.asyncio
class TestCheckpointer:
    """Test PostgreSQL checkpointing"""

    @pytest.fixture
    def checkpointer(self):
        """Create test checkpointer"""
        # Use test database
        conn_str = "postgresql://optiinfra:password@localhost:5432/optiinfra_test"
        return PostgreSQLCheckpointer(conn_str)

    async def test_save_and_load_checkpoint(self, checkpointer):
        """Test checkpoint persistence"""
        workflow_id = "test_wf_123"

        checkpoint = {
            "id": "checkpoint_1",
            "state": {
                "customer_id": "cust_123",
                "workflow_id": workflow_id,
                "workflow_type": "test"
            }
        }

        metadata = {
            "customer_id": "cust_123",
            "workflow_type": "test"
        }

        config = {
```

```
    "configurable": {
        "workflow_id": workflow_id
    }
}

# Save checkpoint
checkpointer.put(config, checkpoint, metadata)

# Load checkpoint
loaded = checkpointer.get(config)

assert loaded is not None
assert loaded["state"]["workflow_id"] == workflow_id
```

📁 FILE STRUCTURE

After implementation, your directory structure should look like:

```
services/cost-agent/
├── src/
│   ├── workflows/
│   │   ├── __init__.py
│   │   ├── states.py          # ✅ NEW: State definitions
│   │   ├── base.py            # ✅ NEW: Base workflow class
│   │   ├── checkpointer.py    # ✅ NEW: PostgreSQL checkpointer
│   │   └── graph_builder.py   # ✅ NEW: Graph utilities
│   ├── api/
│   │   └── routes.py          # Existing
│   ├── collectors/
│   │   └── __init__.py        # Existing
│   └── main.py                # Existing
├── tests/
│   └── test_workflows.py      # ✅ NEW: Workflow tests
├── requirements.txt           # ✅ UPDATED: Added LangGraph deps
└── README.md
```

🔑 KEY CONCEPTS

1. State Management

LangGraph uses typed dictionaries for state:

python

```python
class OptimizationState(TypedDict):
    customer_id: str          # Required
    workflow_id: str          # Required
    recommendations: Optional[List]  # Optional
```

Benefits:

- Type safety
- IDE autocomplete
- Clear contracts

2. Nodes and Edges

Nodes = Functions that transform state:

python

```python
async def analyze(state: OptimizationState) -> OptimizationState:
    # Do work
    state["analysis_results"] = results
    return state
```

Edges = Connections between nodes:

python

```python
graph.add_edge("analyze", "generate_recommendations")
```

3. Conditional Routing

Route based on state:

python

```python
def should_wait_for_approval(state):
    if state["requires_approval"]:
        return "wait_approval"
    return "execute"

graph.add_conditional_edges(
    "check_approval",
    should_wait_for_approval,
    {
        "wait_approval": "wait_approval",
        "execute": "execute"
    }
)
```

4. Checkpointing

Persist state for:

- **Recovery:** Resume after failure
- **Approval:** Pause for human input
- **Audit:** Track all state changes

python

```python
checkpointer = PostgreSQLCheckpointer(conn_string)
graph.compile(checkpointer=checkpointer)
```

🎯 NEXT STEPS

After completing this phase:

1. **Verify:** Run `PHASE1-1.5_PART2_Execution_and_Validation.md`
2. **Next Phase:** PHASE1-1.6 (Spot Migration Workflow)
3. **Integration:** Connect to Orchestrator for agent registration

---

📚 REFERENCES

- **LangGraph Docs:** https://python.langchain.com/docs/langgraph
- **LangChain Core:** https://python.langchain.com/docs/langchain-core
- **StateGraph API:** https://python.langchain.com/docs/langgraph/reference/graphs

---

Document Version: 1.0
Last Updated: October 21, 2025
Status: ✅ Ready for Implementation